CODE PORTABILITY FOR FPGA-BASED SIGNAL PROCESSING SUBSYSTEMS IN SOFTWARE-DEFINED RADIOS

Jim Hwang (Xilinx, Inc., San Jose, CA, 95124, USA, jim.hwang@xilinx.com)

ABSTRACT

Code portability for FPGA-based signal processing is a significant aspect of recent efforts to define a hardware abstraction layer (HAL) for the signal processing subsystems of software-defined radios. In this paper, we show how a platform-based approach to FPGA design can provide an ability to target multiple FPGA families or an ASIC from a single source model. The approach combines direct mapping of a Simulink model with code generation of register-transfer level HDL. We demonstrate that it is possible to generate portable code for DSP systems from Simulink without having to compromise performance of the FPGA realization. This work complements HAL recommendations for portability and (executable) specification by focusing on mechanisms, guidelines, and methodologies for constructing signal processing functions in FPGAs.

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) are widely used to implement physical layer signal processing functions for software-defined radios (SDRs [1] [2]). FPGAs provide very high performance custom hardware solutions, and can be reconfigured in system, and when bringing up a new waveform in the modem. Despite their reprogrammability, they have historically been considered part of the "hardware" within a modem, rather than part of the "software". Consequently, the SDR software control layer, or Software Communications Architecture (SCA[3]), has largely ignored issues related to the specification, configuration, signal transport, or inter-component interfaces that are important to the platform provider of an SDR that exploits FPGAs.

The U.S. government has been a primary driver for SDRs, with significant investment in the technologies and products, e.g., as part of the Joint Tactical Radio System (JTRS) program run by the Joint Program Office (JPO) of the U.S. Department of Defense. Whereas such government programs have lifetimes on the order of ten or more years, FPGA vendors continue to provide new devices roughly every 12-18 months. The increased signal processing capability of new families of FPGAs has remained sufficiently compelling that most platforms require retooling to incorporate new devices as they become available.

Recognizing that the current SCA standard does not sufficiently address the design and deployment of the FPGA portion of the modem, the JPO has recently embarked upon a concentrated effort to extend the SCA to provide guidance and ideally, standardization, for the use of FPGA technologies within SDRs [4]. At roughly the same time, the Software Defined Radio Forum formed a working group devoted to providing recommendations for a hardware abstraction layer to assist in the development, maintenance, and cost management of SDRs [5].

Many viewpoints have been brought forth by design tool, component, and platform vendors, as well as by system integrators and subcontractors for the JTRS program. However, there is general agreement that FPGA code portability is an important, but to date, largely neglected aspect of design methodologies for SDRs.

In this paper, we describe a platform-based approach for obtaining portable FPGA source code, whilst simultaneously providing executable specifications, test harnesses, and "golden" test vectors (i.e., providing accurate input/output relations for establishing conformance to specification through simulation). Our approach treats a high-level system model specified in Simulink [6] as the source code for an FPGA implementation. A block in the model may map onto a set of intellectual property blocks provided by the vendor that exploit vendor-specific device resources to implement the block's function efficiently in a number of FPGA families. Alternatively, a block may map onto a behavioral description in a hardware description language that is inherently portable. It is on the latter case that we focus in this paper. The approach extends widely used FPGA design techniques, using industry standard design tools. Although described in terms of proprietary (though commercially available) tools for Xilinx FPGAs, out approach is equally applicable to other devices.

In Section 2, we present several definitions of code portability, and comment on their feasibility with current device technologies and design tools. Section 3 provides a brief introduction to a platform-based design methodology for implementing DSP systems in FPGAs that underlies our approach to code portability. Section 4 describes a case study of the approach, building a fractionally spaced equalizer (FSE) for a QAM system that is relatively simple, but a representative example of a modem function well-suited to an FPGA.

2. CODE PORTABILITY

All major FPGA vendors have multiple device product lines, each of which is further divided into families that are further divided into part types that differ in available resources, speed grade, and packaging. For example, Xilinx has two primary FPGA product lines: Virtex, which targets highest performance and gate density, and Spartan, which targets high volume and lower cost applications. The most recent families are Virtex-4 and Spartan-3, respectively [7][8].

Because a new FPGA family is introduced roughly every 12-18 months, and the design cycle for a major SDR design can be a significant fraction of this period, the implications of code portability (or more accurately, non-portability) are clear. Often a system must be built to target a family in advance of broadly available silicon.

Bitstream portability means that a bitstream for FPGA family *v(i)* will run directly, possibly via an intermediate run-time software layer, on a *v(i+1)* part. In terms of cost reduction, it is also desirable that a bitstream for FPGA family *v(i)* run directly on a different family device *s(j)*. However, at the current time no FPGA vendor supports bitstream portability.

Source-level portability implies that source code written for device *v(i)* will run after recompilation (but otherwise without change) in device *v(i+1)*. It is desirable to have source level portability between families *v(i)* and *s(i)*. Many FPGA users adopt internal coding guidelines to facilitate full or near source-level portability. In this paper, we describe one way in which source level portability can be achieved using existing devices and design tools.

2.1. Register Transfer Level HDL

The prevailing abstraction in hardware description languages for FPGA design is register transfer level (RTL), which can be synthesized into device-specific logic resources [9]. At this level of abstraction, a design is a network of combinational circuits separated by registers. Registers and other circuit elements are represented behaviorally through idioms inferable by commercial synthesis tools. This style of coding allows the user to specify for example an addition operation with the operator '+', with the synthesis tool mapping this appropriately to device specific architecture primitives.

Considerable progress has been made over recent years in commercial synthesis tools to efficiently target FPGAs. In addition to technology mapping, synthesis tools also apply optimization algorithms to a circuit that preserve behavior, while improving the circuit quality under well-defined criteria (typically logic area or performance). Of particular interest is retiming, which is the reallocation of unit delays (e.g. registers) throughout a circuit, in order to reduce the number of combinational logic levels [9]. There is a close correlation between the largest number of logic levels and the frequency with which bounding registers can be clocked without setup or hold time violations, so retiming is a particularly effective synthesis optimization.

3. A PLATFORM-BASED APPROACH TO FPGA DESIGN

Design methodologies for FPGAs historically lagged those for application specific integrated circuits (ASICs) by roughly a decade, in large part because until recently, the design complexity lagged by roughly the same time. However, as device geometries have continued to shrink, the relative complexity of FPGA designs has increased more rapidly than that of ASICs. Ideas relating to platform-based design, originally motivated by systems-on-chip ASICs [10] have been increasingly adopted for FPGA design [11].

We interpret a platform as an intermediary between abstract behavior and realizable function. Viewed from above, the platform is a restriction on the space of all realizable systems, but one that can be usefully employed to capture the behavior of end applications. Viewed from below, the platform is a restriction on the space of all possible applications, but one that can be readily realized.

More specifically, the platform is a set of arithmetic, logic, memory, and other functional abstractions that allow a user to specify an FPGA-based signal processing subsystem in a natural way. Functions in the platform are chosen so that they can be implemented efficiently, possibly in a number of distinct ways according to additional constraints. As "platform provider", we implement a library of operators, functions, and objects that can be composed within a high-level framework to implement DSP systems. To the application programmer, the library can be used (and extended) to specify a rich set of DSP systems.

In this paper, we address one aspect of platform-based design, namely, how this approach can be used in a commercially available framework to obtain portable, yet highly efficient FPGA code.

3.1 System Generator for DSP

System Generator for DSP is a software framework for modeling and implementing systems in FPGAs using

Simulink [11]. Simulink provides a powerful component-based computing model that is well suited for specifying the concurrency in a custom signal processing architecture. System Generator provides libraries of functions and hardware-related abstractions that can be used to model a signal processing system suitable for FPGAs. Such models are bit and cycle accurate to FPGA hardware. System Generator ensures this by providing automatic code generation from Simulink to a combination of synthesizeable HDL and intellectual property (IP) cores. In addition, System Generator extends Simulink to include event-driven HDL semantics, hardware co-simulation, and rich customization interfaces traditionally associated with modern programming languages [2][13].

In this paper, we focus on an aspect of System Generator that is not widely appreciated: it has the ability to create generic RTL that is extremely efficient, and is portable.

There are three ways to obtain RTL code with System Generator:

- Importing an HDL module using the System Generator Black Box interface. Although a trivial "mapping", this capability is powerful and should not be ignored;
- Using blocks that have RTL implementations, such as the Expression block, Register, Delay, up and down samplers;
- Using the MCode block, which maps MATLAB .m code to synthesizeable VHDL.

Because of its importance and utility, we concentrate on the MCode block and its application.

3.2 System Generator M-Code Block

The System Generator MCode block provides an interface for interpreting a MATLAB .m function in the context of a Simulink simulation. The block is a convenient and flexible way to realize arithmetic functions as well as finite state machines and control logic in the context of System Generator. In contrast with the Simulink S-function API [6], the MCode block simply interprets m-code as its input-output relation.

The block accepts an m-code function as a mask parameter, and adapts its interface to that of the function. A function argument can be treated either as an input port or as a parameter internal to the function (i.e. run-time constant), under the control of a block mask parameter. Return values are interpreted as output ports on the block. The m-code is translated in a straightforward way into equivalent behavioral VHDL during code generation.

In the System Generator v6.3 release, the MCode block supports combinational functions and functions with internal state. Language constructs include nested branches (switch, if/then/else), assignment, arithmetic operators (+, *), bit wise logical, and a number of other operators [12]. System Generator provides MATLAB and Simulink-based fixed-point data types (prior to the MATLAB R14 release, there was no fixed-point type available in MATLAB).

The MCode block automatically infers a lossless type for internal variables and return types, based on the input types. In addition, the block performs dead code elimination and other optimizations during code generation. By specifying m-code function arguments as internal parameters, it is straightforward to create parametric blocks that map very efficiently onto hardware.

The mapping from m-code into hardware uses well-established rules (e.g., [14]). As a simple example, a 2-to-1 multiplexer is realized with the following m-function.

```
function [c] = mux2to1(a, b sel)
if (sel == 0), c = a;
else, c = b, end
```

The output type is the smallest container necessary to represent inputs `a` and `b` after binary point alignment. If the select signal is known at compile time, then declaring the `sel` input an internal parameter within the block mask directs System Generator to realize the function in hardware as a wire tied from the appropriate input port. This of course is a particularly simple example of dead-code elimination.

4. CASE STUDY: ADAPTIVE EQUALIZER

Often the flexibility attained by using high-level abstractions comes at a cost of efficiency in the resulting hardware realization. In this section, we demonstrate that this is not always the case. We employ the MCode block to convert a System Generator model that implements a fractionally spaced equalizer (FSE), originally designed to map onto Xilinx IP cores, into an equivalent model that is implemented entirely as RTL VHDL, automatically generated by System Generator. What is perhaps most interesting, is that when the RTL design is mapped using the most recent (as of this writing) version of synthesis tools that incorporate retiming, it achieves a nearly 20% increase in achievable clock rate over the original design.

4.1 FSE Model

The T/2 adaptive FSE has been designed for a 16-QAM modulation system, sampling an input data stream twice per symbol [15]. The equalizer consists of three modules: an 8-tap complex LMS filter implemented with a two-way polyphase decomposition, a symbol demapper (also used to generate the "desired" signal for the LMS update), and the LMS update [3].

The original System Generator model for the T/2 adaptive FSE has been included as a demonstration design since the System Generator v3.1 release. In the v6.3 release, both the original and fully synthesizeable models are provided. It was not necessary to modify the structure of the design in any fundamental way to derive a fully synthesizeable model. The hierarchy was preserved, and for the most part, only leaf nodes needed replacement by equivalent MCode blocks.

The LMS filter is constructed of low level blocks, including adder/subtractors, accumulators, counters, multiplexers, multipliers, up- and down-samplers, and simple memory elements. Delay blocks in the original model that employed SRL16 resources (shift register logic unique to Xilinx FPGAs) were replaced by generic register-based delay lines. M-functions defining adder/subtractor, multiplexer, and multipliers were simply instrumented. Counters and accumulators were constructed using the adder, delay, and constant blocks as Simulink subsystems.

Several subsystems in the original model were realized as straightforward transcriptions into m-code. For example, the symbol demapper shown in Fig. 1 was replaced by a MCode block with the MATLAB function shown below.

```
function [v] = QAM4map(i)

% type declarations
utype = {xlUnsigned,8,8,xlRound, xlSaturate};
stype = {xlSigned,8, 8, xlRound, xlSaturate};
rtype = {xlSigned,10,8, xlRound, xlSaturate};
%symbolic constants
two3rds = xfix(utype, 2/3);
one3rd  = xfix(stype, 1/3);
neg3rd  = xfix(stype, -1/3);
% state variables (pipeline latency = 2)
persistent r0, r0 = xl_state(0, rtype);
persistent r1, r1 = xl_state(0, rtype);

v  = r1, r1 = r0;
msb= xl_slice(i,xl_nbits(i)-1,xl_nbits(i)-1);
if (msb == 1)
  if (two3rds < -i), r0 = -1;
  else, r0 = neg3rd;
  end
else
  if (two3rds < i),  r0 = 1;
  else, r0 = one3rd;
  end
end
```

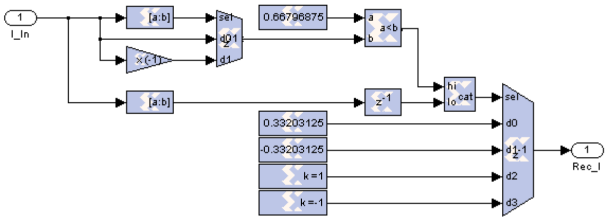

Figure 1. QAM Symbol Demapper

4.2 Implementation Results

The original and synthesizeable designs were built using System Generator v6.2 software, and each synthesized for a V-II Pro (-7 speed grade) and Spartan-3 (-5 speed grade) FPGA using Synplify Pro 7.6, with retiming and pipelining options enabled. The designs were run through mapping, placement, and routing using the Xilinx ISE 6.2.02i software, with highest placer and router effort levels. This process was run repeatedly with different clock frequency constraints in order to determine the highest frequency obtainable. The results, summarized in the following Table are somewhat surprising.

Part Type	MHz Cores	MHz Synth	LUTs Cores	LUTs Synth	DFFs Cores	DFF Synth
xc2vp20	88.9	104.1	1636	1595	1868	2195
xc3s150 0	79.3	85.5	1636	1531	1868	2193

Although one might expect the generic RTL implementation to run at a lower clock frequency than the original that employed IP cores, in fact the reverse was true. For both Virtex-II Pro and Spartan-3 devices, the RTL version ran at an appreciably higher clock frequency. The RTL implementations used more DFF registers than the core-based implementations (recall the MCode did not specify SRL16 resources, although the synthesis tool was free to map onto them when it could infer them correctly).

Current synthesis tools treat IP cores as black boxes, and no optimizations are available that cross module boundaries. The RTL design in contrast allowed the synthesis tool to freely move registers and optimize logic across module boundaries. It should be noted however, that in all prior versions of the Synplify Pro (as well as all versions of the Xilinx XST tool), retiming did not provide significant speed-up for this design. One concludes that logic synthesis optimizations necessary for high-performance, portable design are still in an early stage of development.

5. CONCLUSIONS

With the increased adoption of FPGAs as signal processors comes an increased expectation for design flows and methodologies that support programming models similar to those for general purpose and DSP processors. Code portability, at least at the source level (i.e., admitting recompilation) is of fundamental importance. Although FPGA source code is not as widely portable as code for general-purpose microprocessors, we have demonstrated how System Generator and similar design tools provide considerable progress towards this end. Using an adaptive FSE as an example, we have shown how a single System Generator model can be used to specify both behavior and implementation, producing a generic RTL implementation suitable for an FPGA. The design exploits retiming and logic synthesis optimizations in order to achieve high performance.

6. ACKNOWLEDGEMENTS

The author gratefully acknowledges the contributions of, and numerous fruitful discussions with colleagues Brent Milne, Haibing Ma, and Brad Taylor.

7. REFERENCES

[1] Berkeley Design Technology, Inc. "FPGAs for DSP," Focus Report, July 2002, http://www.bdti.com/products/reports_focus.html

[2] C. Dick and J. Hwang, "FPGAs: A Platform-Based Approach to Software Radios," in *Software Defined Radio: Baseband Technologies for 3G Handsets and Basestations,* (W.H.W. Tuttlebee, Editor), Wiley, 2004.

[3] U.S. Department of Defense, Joint Tactical Radio System, Software Communication Architecture, Technical Overview. http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html

[4] U.S. Department of Defense JTRS Joint Program Office, SCA Extensions Workshop, Arlington, Va., April 2004, http://jtrs.army.mil/sections/programinfo/fset_programinfo.html?programinfo_industry.

[5] Software Defined Radio Forum, Hardware Abstraction Working Group, http://www.sdrforum.org/tech_comm/halwg.html.

[6] The Mathworks, Inc., *Using Simulink*, 2002.

[7] Xilinx, Inc., *Virtex-4 Handbook*, http://www.xilinx.com/products/virtex4/download.htm

[8] Xilinx, Inc., *Spartan-3 Data Sheet,* http://direct.xilinx.com/bvdocs/publications/ds099.pdf.

[9] G. DiMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[10] K. Keutzer, A.R. Newton, J.M Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: orthogonalization of concerns and platform-based design", *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp.1523-43, 2000.

[11] J. Hwang, B. Milne, N. Shirazi, J. Stroomer, "System Level Tools for FPGAs," *Proceedings FPL 2001*. Springer-Verlag 2001.

[12] Xilinx, Inc., *System Generator for DSP User Guide*, http://www.support.xilinx.com/products/software/sysgen/app_docs/user_guide.htm.

[13] V. Singh, A. Root, E. Hemphill, N. Shirazi, J. Hwang, "Accelerating a Bit Error Rate Tester with a System Level Tool," F*ield-Programmable Custom Computing Machines, FCCM 2003*, Proceedings, IEEE 2003.

[14] I. Page and W. Luk, "Compiling occam into FPGAs*", in FPGAs, W. Moore and W. Luk (editors)*, Abingdon EE&CS Books, 1991, pp. 271-283.

[15] J.R. Treichler, I. Fijalkow, and C.R. Johnson, Jr., "Fractionally Spaced Equalizers", *IEEE Signal Processing Magazine*, May 1996, pp. 65-81

[16] C.Dick, "Design and implementation of high-performance FPGA signal processing datapaths for software-defined radios", *VMEbus Systems*, August 2001.